An ASM Definition of the Dynamic OCL 2.0 Semantics

Stephan Flake[1] and Wolfgang Mueller[2]

[1] ORGA Systems GmbH
Am Hoppenhof 33, 33104 Paderborn, Germany

[2] Paderborn University / C-LAB
Fürstenallee 11, 33102 Paderborn, Germany

Abstract. The recently adopted OCL 2.0 specification comes with a formal semantics that is based on set theory with a notion of an object model and system states. System states keep the runtime information relevant for the evaluation of OCL expressions. However, not all new language concepts of OCL 2.0 are already addressed in that formal semantics. We show how to overcome this by introducing new components to the object model and system states defining a dynamic semantics of OCL. In order to give precise rules that determine when the current system state has to be updated according to a change in the referred UML model, we make use of adequate mathematical means, namely Abstract State Machines (ASMs). Though our ASM specification also gives a clear definition for the evaluation of OCL constraints, it leaves sufficient flexibility for application specific implementations that have to determine when constraints are to be checked.

1 Introduction

The recently adopted OCL 2.0 specification provides both a metamodel-based as well as a formal semantics definition [11]. The formal semantics is based on set theory with the notion of an object model, which is basically a formalization of UML Class Diagrams. An instantiation of an object model is called a *system*. A system changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during system execution. The information that is needed to evaluate OCL expressions is stored in *system states*, which represent snapshots of the running system. For the evaluation of OCL expressions, the adopted OCL 2.0 specification provides a denotational semantics by interpretation functions over *environments*, i.e., tuples of system states and OCL-specific variable assignments.

Several new language concepts have been introduced to OCL 2.0, like tuples, messages sent, and ordered sets. However, not all of the new concepts are already addressed in the formal semantics. In particular, the formal OCL 2.0 semantics currently lacks a formalization of

- operations on predefined collection type `OrderedSet`,
- global variable definitions, called *def-clauses*[1],

[1] Leaving out this language concept leads to a significant loss in the expressiveness of OCL. See [4] for more details.

– operations on State Diagram states, and
– operators to access OCL messages and operations to reason about them.

Note that operations defined for ordered sets are basically the same as for sequences and that def-clauses are mapped to so-called `OclHelper` variables and operations [11, Section 7.4.4]. `OclHelper` variables and operations, in turn, are stereotyped attributes and operations of classifiers. Such variables and operations can be used in OCL expressions just like common attributes and operations. Thus, it only has to be ensured that no naming conflicts occur.

We already integrated UML State Diagrams to OCL and defined a formal semantics for the predefined operation `oclInState()` [6] and also formally defined operators and operations for OCL messages [5]. These definitions are based on extensions of the work by Richters which heavily influenced the formal semantics of OCL 2.0 [13]. We introduced additional components to the object model and system states and could then give a denotational semantics by interpretation functions for the predefined OCL operations that are still missing in the formal semantics of OCL 2.0, e.g., the message-related operations `hasReturned()` and `result()`.

This extension is also the foundation for defining a *dynamic semantics* of OCL, which is in the focus of this article. Note that the (extended) denotational semantics allow to evaluate OCL 2.0 expressions over given system states, but there are no precise rules that determine *how* system states have to be updated in relation to the execution of the referred UML model. To overcome this, we make use of adequate mathematical means for state-oriented operational definitions, namely Abstract State Machines (ASMs).

ASMs were introduced by Gurevich [7]. Based on the notion of a virtual machine execution in combination with a mathematically precise notion of states and state transitions known as *algebras*, they provide a concise and rigorous but yet intuitive way to define system semantics. ASMs are well-established in the domain of formal specification and have already been successfully applied to define the semantics of, e.g., UML Activity Diagrams [1] and the run-to-completion step of UML State Diagrams [2].

The remainder of this article is structured as follows. The next section briefly outlines the basics of ASMs. In Section 3, we present the extensions to the object model and system states that are necessary to be able to evaluate OCL expressions that also make use of state- and message-related operations. Section 4 then presents the dynamic semantics of OCL with ASMs. In Section 5, we briefly discuss related work. Section 6 closes with a conclusion.

2 Abstract State Machines

Abstract State Machine (ASM) specifications can be understood as *pseudocode over abstract data* without any particular theoretical prerequisites. We here only list the basic definitions and refer to [7, 3] for a formal introduction and more details. An ASM specification comes in form of guarded function updates, called *rules*, of the form

if $Condition$ **then** $<Updates>$ **else** $<Updates>$ **endif**

Rules are presented as nested if-then-else clauses with a set of function updates in their body. When executing the rules, the underlying ASM abstract machine executes state transitions with *algebras* as states. An algebra can be seen as a *database of functions* [3]. Basically, an algebra is a mathematical structure over abstract objects that are elements of a domain (or: universe). A particular function or relation for an object obj is described by a parameterized function f, which assigns to each x the value $f(obj, x)$. Partial functions are turned into total functions by setting $f(obj, x) = undef$, where $undef$ is a special predefined value denoting that $f(obj, x)$ is undefined. Note that 0-ary functions play the role of variables known from imperative programming languages.

Functions have a well-defined signature and mapping. ASMs distinguish static and dynamic functions. Static functions do not change during executions of the ASM, i.e., the function values do not depend on the states of the ASM. In contrast, the values of dynamic functions might change, either because of an update in the ASM itself or by the environment. ASMs distinguish between four kinds of dynamic functions, i.e., *controlled*, *monitored*, *interaction*, and *out* functions. Controlled functions can only be read and changed by the ASM itself, while monitored functions can only be read by the ASM and are changed by the environment. Interaction (or shared) functions can be changed by both the ASM and the environment, but then some mechanism is necessary to guarantee consistency. Finally, out functions are changed but cannot be read by the ASM.

Firing a set of rules in one step performs a *state transition*. Only those rules are fired whose guards (i.e., *Condition*) evaluate to true. At each step, the guards evaluate to a set of *function updates*, each of the form $f(t_1, ..., t_r) := t_0$, where t_i are terms (including functions). A *block* is a set of function updates which we separate by commas. The individual function updates of each block are collected in a so-called *update set* and are simultaneously executed in the same step. Each function update changes a value at a specific *location* that is given by the left hand side of the update. Functions are considered to be global, such that two or more simultaneous updates of the same location in one update set define inconsistency.

In the case of inconsistency no state transition is performed and no update is being executed.

We demonstrate a simple guarded update by the following example:

$$\textbf{if } true \textbf{ then } A := B, B := A \textbf{ endif}$$

That definition gives an simultaneous update of the 0-ary functions A and B. Since both updates are simultaneously executed, the values are swapped (A becomes the value of B and vice versa). Due to its true condition, the rule fires at each step.

ASMs are multi-sorted based on the notion of universes. We presume the standard mathematic universes of booleans, integers, lists, etc. as well as the standard operations on them without further mention.

The **choose** constructor defines an arbitrary selection of one element in a universe

$$\textbf{choose } v \textbf{ in } Universe \ <Rule> \ \textbf{endchoose}$$

where v is (non-deterministically) selected from the given universe. The **choose** constructor can be qualified by an additional condition indicated by the keyword **satisfying**.

The **var** rule constructor defines the simultaneous instantiation of a rule:

$$\textbf{var } v \textbf{ ranges over } Universe \ <Rule> \textbf{ endvar}$$

Executing the constructor means to execute the rule for each element in $Universe$ simultaneously, i.e., the constructor basically spawns n rules where n is the number of elements in $Universe$.

3 Extended Object Model and System States

The formal definition of the object model in OCL 2.0 is based on the object model of Richters [13]. However, this formalization lacks some of the new OCL 2.0 language concepts. We therefore define an extension of the object model called *extended object model*, in which a number of concepts are newly introduced (cf. Section 3.1). Correspondingly, additional information has to be stored in system states to be able to evaluate OCL 2.0 expressions (cf. Section 3.2). The completed system state description then allows to define a high-level dynamic semantics for OCL by means of *traces* (cf. Section 4).

3.1 Syntax

In the remainder of this article, let $\mathcal{A}$ be an alphabet, $\mathcal{N}$ be a set of names over $\mathcal{A}^+$, and T a set of types. In particular, $T = T_B \cup T_E \cup T_C \cup T_S$ comprises a set of basic standard library types T_B, i.e., $Integer$, $Real$, $Boolean$, and $String$, a set T_E of user-defined enumeration types, a set T_C of user-defined classes, $c \in CLASS$, and a set of special types $T_S = \{OclVoid, OclState, OclAny\}$.

We call the value set $I(t)$ represented by a type t the *type domain*. For convenience, we presume that `OclUndefined` (in the following denoted by symbol $\perp$) is included in each type domain, such that we have, e.g., $I(OclVoid) = \{\perp\}$ and $I(OclAny) = \bigcup_{t \in T_B \cup T_E \cup T_C \cup \{OclState\}} I(t)$.

Furthermore, let $c \in CLASS$ be a class and $t_c \in T_C$ be the type of class c.[2] Each class c has a set ATT_c of attributes that describe characteristics of their objects. An attribute has a name $a \in \mathcal{N}$ and a type $t \in T$ that specifies the domain of attribute values. A class c is also associated with a set OP_c of operations and a set SIG_c of signals.

We define the *Extended Object Model* $\mathcal{M}$ by the tuple

$$\mathcal{M} = \langle \begin{array}{l} CLASS, ATT, OP, paramKind, isQuery, SIG, SC, \\ ASSOC, \prec, \prec_{sig}, associates, roles, multiplicities \end{array} \rangle$$

with

- a set $CLASS = ACTIVE \cup PASSIVE$ of active and passive classes,
- a set of attributes, $ATT = \bigcup_{c \in CLASS} ATT_c$,

[2] Each class $c \in CLASS$ induces an object type $t_c \in T$ that has the same name as the class. The difference between c and t_c is that we have the special value $\perp \in I(t_c)$ for all $c \in CLASS$.

– a set OP of operations, $OP = \bigcup_{c \in CLASS} OP_c$,
– a function $paramKind : CLASS \times OP \times \mathbb{N} \to \{in, inout, out\}$ that gives for each operation parameter its parameter kind,
– a function $isQuery : CLASS \times OP \to Boolean$ that determines whether an operation is a query operation or not,
– a set SIG of signals, $SIG \supseteq \bigcup_{c \in CLASS} SIG_c$,
– a set SC of State Diagrams (or: StateCharts), $SC = \bigcup_{c \in ACTIVE} SC_c$,
– a set $ASSOC$ of associations between classes,
– generalization hierarchies $\prec$ for classes and $\prec_{sig}$ for signals, and
– functions $associates$, $roles$, and $multiplicities$ that define a mapping for each element in $ASSOC$ to the participating classes, their corresponding role names, and multiplicities, respectively.

That definition should be sufficient for the remainder of this article. For more details about sets $CLASS$, ATT, OP, and $ASSOC$, readers are referred to the corresponding sources [13, 11]. We also omit the formal syntax definitions for signals and State Diagrams and refer to [6] for further details. Concerning State Diagrams and their inheritance among classes, we assume that they comply to some *inheritance policy*. Though the UML standard suggests some informal policies [12, Section 2.12.5], different other formal notions for behavioral consistency have been identified in the literature, e.g., [14].

The set of characteristics defined in a class together with the inherited characteristics is called the *full descriptor of a class*. Formally, this is a tuple

$$FD_c = \langle\ ATT_c^*, OP_c^*, paramKind_c^*, isQuery_c^*, SIG_c^*, SC_c, navEnds^*(c)\ \rangle$$

with the complete sets of attributes, operations, signals, navigable role names, and – in the case of an active class – the associated State Diagram. For example, the complete set of attributes of a class c is defined by

$$ATT_c^* = ATT_c \cup \bigcup_{c' \in parents(c)} ATT_{c'},$$

where $parents(c)$ denotes the set of (transitive) superclasses of c. The complete sets OP_c^*, SIG_c^*, and $navEnds^*(c)$ of operations, signals, and navigable role names are defined correspondingly. Functions $isQuery_c^* : OP_c^* \to Boolean$ and $paramKind_c^* : OP_c^* \times \mathbb{N} \to \{in, inout, out\}$ are derived from functions $isQuery$ and $paramKind$, respectively.

3.2 System State

The domain $I_{CLASS}(c)$ of a class $c \in CLASS$ is the set of objects of class c and all of its child classes. For technical purposes, we define $I_{CLASS} = \bigcup_{c' \in CLASS} I_{CLASS}(c')$. Objects are referred to by object identifiers that are unique in the context of the whole system. The set of object identifiers of a class $c \in CLASS$ is defined by an infinite set $oid(c) = \{oid_1, oid_2, \ldots\}$.

Note that – in contrast to the current OCL semantics – we distinguish between 'real' objects $\underline{oid}$ and their identifiers oid in the remainder of this article, simply by using underlines.

The current notion of a system state with only three components (i.e., current objects, their attribute values, and the established links) is not sufficient to be able to evaluate OCL 2.0 expressions. Additionally, we need information about currently activated states, operations called, signals sent, currently executed operations, etc. In this context, we adopt ideas of Ziemann and Gogolla [16] to formalize currently executed operations and define further functions to capture the required additional information. Formally, a *system state* over the extended object model $\mathcal{M}$ is a tuple

$$\sigma(\mathcal{M}) = \langle\ \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF}, \Sigma_{currentOp}, \Sigma_{currentOpParam}, \Sigma_{sentMsg}, \Sigma_{sentMsgParam}, \Sigma_{inputQueue}, \Sigma_{inputQueueParam}\ \rangle\ .$$

We explain the components of system states in more detail, but note that Σ_{CLASS}, Σ_{ATT}, and Σ_{ASSOC} are already defined in [13, 11].

(1) $\Sigma_{CLASS} = \bigcup_{c \in CLASS}\ \sigma_{CLASS}(c)$. The finite sets $\sigma_{CLASS}(c)$ comprise all currently existing objects of class c, i.e., $\sigma_{CLASS}(c) \subseteq oid(c) \subseteq I_{CLASS}(c)$. For further application, we define $\sigma_{ACTIVE}(c)$ for active classes correspondingly.

(2) The current attribute values are kept in set Σ_{ATT}. It is the union of functions $\sigma_{ATT}(a) : \sigma_{CLASS}(c) \rightarrow I(t)$, where $a \in ATT_c$ and t is the type specified for a. Each function $\sigma_{ATT}(a)$ assigns a value to attribute a for each currently existing object of class c.

(3) $\Sigma_{ASSOC} = \bigcup_{as \in ASSOC}\ \sigma_{ASSOC}(as)$ comprises the finite sets $\sigma_{ASSOC}(as)$ that contain links that connect objects. We refer to the sources mentioned above for detailed information about links.

(4) The current State Diagram configurations are kept in set

$$\Sigma_{CONF} = \bigcup_{c \in ACTIVE}\ \{\ \sigma_{CONF}(c) : \sigma_{ACTIVE}(c) \rightarrow I_{SC}(c)\ \}\ .$$

Each function $\sigma_{CONF}(c)$ assigns an active state configuration to each object of a given class $c \in ACTIVE$. Set $I_{SC}(c)$ denotes the set of valid state configurations of the State Diagram SC_c. For a formal definition of state configurations, see [6].

The following subsections describe the new system state components that relate to the *local snapshots* of the metamodel-based semantics of OCL 2.0 [11, Section 10.2.1].

Currently Executed Operations. Let $\mathcal{ID}$ be an infinite enumerable set, e.g., $\mathcal{ID} = \mathbb{N}$, and let $Status = \{executing, returning\}$. At the starting point of an operation execution, a unique identifier $opId \in \mathcal{ID}$ is associated with the current operation execution. Thus, an operation execution can uniquely be identified by a given object identifier, an operation signature $op \in OP$, and an operation identifier $opId \in \mathcal{ID}$. The set of currently executed operations is defined by

$$\Sigma_{currentOp} = \bigcup_{c \in CLASS}\ \{\ \sigma_{currentOp,c} : \sigma_{CLASS}(c) \times OP_c^* \rightarrow \mathcal{P}(I_{CLASS} \times OP \times \mathcal{ID} \times \mathcal{ID} \times Status)\ \}\ .$$

Each function $\sigma_{currentOp,c}$ gives a set of tuples of the form $\langle srcId, srcOp, srcOpId, opId, status\rangle$ that uniquely identify all currently executed operations for a given object and operation name. Elements $srcId$, $srcOp$, and $srcOpId$ refer to the operation execution that originally invoked the considered operation op with identifier $opId$. These elements are necessary to have a reference for returning a result value after operation termination.

A flag $\in Status$ indicates the current status of operation execution. Compared to the messaging actions specified in UML 1.5, we here omit statuses $ready$ and $complete$ [12, Section 2.19.2.3], as they are not necessary in the context of OCL.

Actual parameter values of executed operations are kept in $\Sigma_{currentOpParam} =$

$$\bigcup_{c\in CLASS} \left\{ \begin{array}{l} \sigma_{currentOpParam,c} : \\ \quad \sigma_{CLASS}(c) \times OP_c^* \times \mathcal{ID} \rightarrow I^?(t_1) \times ... \times I^?(t_n) \times I^?(t) \end{array} \right\}.$$

Each function $\sigma_{currentOpParam,c}$ gives the actual parameter values of the currently executed operations. In the definition above, we applied sets $I^?(t) = I(t) \cup \{?\}$ for any $t \in T$. Symbol ? denotes the ***unspecified status*** of a value. This symbol must not be mixed up with the ***undefined value*** denoted by $\perp$ (or `OclUndefined` in the concrete OCL syntax) and is also different from the String literal `'?'`. Only operation parameters i with $paramKind(c, op, i) = out$ and the return value carry the unspecified value during operation execution.

Messages Sent. To be able to evaluate OCL expressions that reason about messages, we have to store the ***history of messages sent*** for each executed operation. For each object $\underline{oid} \in \sigma_{CLASS}(c)$ and each of its currently executed operations op with identifier $opId$, we define a function $\sigma_{sentMsg,c}(\underline{oid}, op, opId)$ that gives the set of messages sent with their corresponding destination objects. We then define $\Sigma_{sentMsg} =$

$$\bigcup_{c\in CLASS} \left\{ \begin{array}{l} \sigma_{sentMsg,c} : \\ \quad \sigma_{CLASS}(c) \times OP_c^* \times \mathcal{ID} \rightarrow \mathcal{P}(I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID}) \end{array} \right\}.$$

Set $\mathcal{ID}$ in $\mathcal{P}(I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID})$ is used to refer to the correct message identifier when returning a value for synchronous operation calls. We here require a total order for $\mathcal{ID}$, such that it is possible to uniquely build sequences of messages sent.

An element $\langle destId, msg, callId\rangle \in \sigma_{sentMsg,c}(\underline{oid}, op, opId)$ denotes that a message with signature msg and call identifier $callId$ has been sent from object $\underline{oid}$ to the (not necessarily still existing) object with identifier $destId$ as part of operation execution op with identifier $opId$.

Additionally, we have to store the actual parameter values of each message sent. The formal definition of functions $\sigma_{sentMsgParam,c}$ is very similar to the definition of the functions for parameters of currently executed operations presented before. We therefore omit further descriptions here and refer to [5] for more details.

Input Queues. Set $\Sigma_{inputQueue}$ is used to store events, i.e., operation calls and signals that are sent to objects and still waiting to be dispatched. While other events like *change events*, *time events*, and implicit *completion events* invoked by (an implementation of) a State Diagram have to be considered in a general notion of an input queue, it is sufficient for us to consider only those events that are relevant for the evaluation of OCL

expressions. We later refer to input queues to update the system state when a signal or operation is dispatched. This enables us to change the set of currently executed operations accordingly, which is essential for a well-defined semantics of OCL message operations. Formally, we have $\Sigma_{inputQueue} =$

$$\bigcup_{c \in CLASS} \{\ \sigma_{inputQueue,c} : \sigma_{CLASS}(c) \rightarrow \mathcal{P}(I_{CLASS} \times OP \times \mathcal{ID} \times\ (SIG_c^* \cup OP_c^*) \times \mathcal{ID})\ \},$$

where each function $\sigma_{inputQueue,c}$ maps to a set of sent signals and operations. The actual parameter values of waiting messages are kept in set $\Sigma_{inputQueueParam}$, and again we omit a formalization here for the sake of conciseness.

We now have all necessary components defined to evaluate general OCL 2.0 expressions and we refer to [6, 5] for the formal semantics of the predefined state- and message-related operations. In the remainder of this article, we make use of the presented extended object model and system states for an ASM definition of the dynamic semantics of OCL.

4 ASM Definition of the Dynamic OCL Semantics

In the simplest case, i.e., when (the implementation of) the system is executed on a single CPU, there is a clear temporal order on the system execution. But when the system is distributed, there is a partial order among the system execution. This problem can be treated in an ideal case by introducing a *global clock* that allows for a global view on the system. For the evaluation of OCL constraints, we assume that we have this global view on the system.

The basic idea of our approach is that the system states are stored such that it is possible to access them at a later point of time. There are two reasons for this approach, both in the context of postcondition evaluation. Firstly, it is possible in postconditions to refer to values at the precondition time of the corresponding operation execution. Secondly, the sequence of messages sent during an operation execution has to be stored to be able to evaluate message-related operations in postconditions.

To record the system changes we are interested in w.r.t. OCL constraints, we first identify the set of *noteworthy changes* which may affect the evaluation of OCL expressions. Each time such a noteworthy change occurs, a new system state is built and appended to the current sequence of system states. This sequence is also called a *trace* in the remainder. A *trace* for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states, i.e.,

$$trace(\mathcal{M}) = \langle\langle\, \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots \rangle\rangle.$$

The first trace element $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state in which all components are empty. Given a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, the next system state $\sigma(\mathcal{M})_{[i+1]}$ is added to the trace when at least one *noteworthy change* occurs. The particular noteworthy changes are further explained in the rule *updateSystemState* of the ASM definition in Subsection 4.2.

4.1 When to Check Constraints

We take a general approach and abstract from the fact *when* OCL constraints are to be evaluated and *what* to do in the case of a constraint failure. This is completely in sense of the OCL developers [15], as they state at the beginning of the corresponding Section 4.6 (page 90):

> *When implementing constraints, you must decide when to check them and what to do when a constraint fails.*

We make use of the monitored ASM function *evaluateConstraints* of type Boolean to trigger the evaluation of OCL constraints. As explained before, it is out of the scope of this definition when this function actually becomes true.

We define $INV = \bigcup_{c \in CLASS} inv(c)$ as the set of all invariants, where $inv(c)$ denotes the set of invariants over class c of the referred UML model. Let $inv^*(c)$ be the full set of invariants for a given class c, i.e., $inv^*(c) = inv(c) \cup \bigcup_{c' \in parents(c)} inv(c')$. Similarly, let PRE and $POST$ represent the sets of all pre- and postconditions, respectively.

The three monitored functions $checkInv : \Sigma_{CLASS} \rightarrow \mathcal{P}(INV)$, $checkPre : \Sigma_{currentOp} \rightarrow \mathcal{P}(PRE)$, and $checkPost : \Sigma_{currentOp} \rightarrow \mathcal{P}(POST)$ provide the invariants, pre-, and postconditions that have to be checked over particular objects and operations when *evaluateConstraints* becomes true. In the remainder, we may also write $\langle obj, inv \rangle \in checkInv$ to refer to an invariant $inv \in checkInv(obj)$. The same holds for pre- and postconditions $\langle opExec, pre \rangle \in checkPre$ and $\langle opExec, post \rangle \in checkPost$. Of course, we require that the constraints are well-defined for the objects and operations, e.g., for all $\langle obj, inv \rangle \in checkInv$ holds that $obj \in \Sigma_{CLASS}(c)$ implies $inv \in inv^*(c)$. Similar restrictions apply for pre- and postconditions.

4.2 ASM Rules

The left hand side of Figure 1 illustrates the general approach of the ASM definition for the dynamic OCL semantics. On the right hand side, the figure gives the corresponding sequential ASM steps until the system execution is stopped. The individual steps are given as ASM *macro* definitions. Macros are placeholders for ASM rules in order to achieve a better readability of the ASM specification.

In state *updateSystemState*, the OCL evaluation continuously updates the system state until *evaluateConstraints* becomes true. When *evaluateConstraints* is true, the evaluation of constraints is started with an initialization. In the next phase, the constraints under investigation, i.e., the elements of $checkInv$, $checkPre$, and $checkPost$, are checked. A Boolean function $violation$ is set to `true` if at least one of the considered constraints is violated. Note here that not only an evaluation to `false`, but also an evaluation to the undefined value $\perp$ is a violation.

In the remainder, we provide the ASM rules that reflect the OCL evaluation cycle of Figure 1. We follow a state-based definition of the individual steps. The individual states

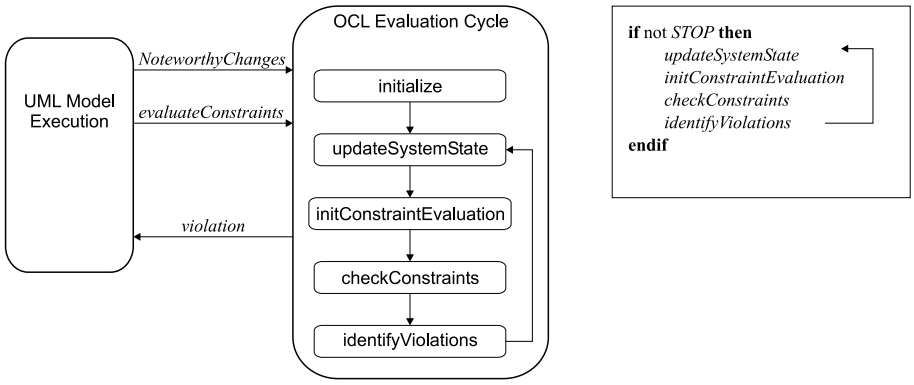

Fig. 1. Overview of the OCL Evaluation Cycle

are represented by the variable *phase* which is checked by each rule and is thereafter set to the corresponding next state.

Initialization. Initially, *phase* is set to *initialize* and *TRACE* is an empty sequence. In the following rule, setting $\sigma(\mathcal{M})_{[0]}$ to $\langle \emptyset, \ldots, \emptyset \rangle$ means that all tuple components are initially empty.

if $phase = initialize$ **then**
  $i := 0$,
  $TRACE.append(\sigma(\mathcal{M})_{[0]} := \langle \emptyset, \ldots, \emptyset \rangle)$,
  $phase := updateSystemState$
endif

Update System State. Once entering phase *updateSystemState*, we continuously check for noteworthy changes in the running system. The kinds of noteworthy changes are listed in Table 1.

Note that different kinds of noteworthy changes might occur in parallel at the same instant of time, such that several of the macros have to be executed simultaneously and build a new system state $\sigma(\mathcal{M})_{[i+1]}$. For example, a number of objects can be created at the same time on different nodes in a distributed system, and in addition one or more new links can be established.

if $noteworthyChange = true \ \wedge \ phase = updateSystemState$ **then**
  $UpdateClasses$, $UpdateAttributes$, $UpdateLinks$,
  $UpdateConfigurations$, $UpdateSignals$, $UpdateOperations$,
  $UpdateInputQueues$, $UpdateMessagesSent$,
  $TRACE.append(\sigma(\mathcal{M})_{[i+1]})$,
  $i := i + 1$,
  if $evaluateConstraints = true$ **then**
    $phase := initConstraintEvaluation$
  endif
endif

In the preceding rule, condition $noteworthyChange$ is defined by

$$noteworthyChange \equiv \bigvee_{X \in Changes} X \neq \emptyset,$$

where the elements of $Changes$ are the sets of Table 1, i.e., $NewObjects$, $DestroyedObjects$, ..., $ReturnedOperations$.

The macros with the prefix *Update* in the preceding rule are for building the new system state $\sigma(\mathcal{M})_{[i+1]}$ w.r.t. the sets of current objects, attribute values, links, configurations, received signals, currently executed operations, and sent messages. All of these update rules are very similar and explained in more detail in [5], but it is sufficient to give just an example rule here to understand the general idea, i.e., the functions of the previous system state are copied and updated corresponding to the given changes in terms of Table 1.

$$\begin{array}{l} UpdateMessagesSent \equiv \\ \quad \forall j \in \{1, \ldots, r\} : \\ \quad\quad \sigma_{sentMsg,c_j}(\underline{oid}_j, msg_j, msgId_j)_{[i+1]} := \\ \quad\quad\quad \sigma_{sentMsg,c_j}(\underline{oid}_j, msg_j, msgId_j)_{[i]} \cup \{\langle destId_j, op_j, callId_j \rangle\}, \\ \quad\quad \sigma_{sentMsgParam,c_j}(\underline{oid}_j, op_j, opId_j, destId_j, msg_j, callId_j)_{[i+1]} := \\ \quad\quad\quad \langle v_{j,1}, ..., v_{j,n_j}, ? \rangle, \end{array}$$

Constraint Evaluation. In the phase *initConstraintEvaluation*, we initialize some help variables that are for documenting potential constraint violations and set *phase* to the next step.

if $phase = initConstraintEvaluation$ **then**
$violation := false$,
$violatedConstraints := \emptyset$,
$undefinedConstraints := \emptyset$,
$UpdateOclVariableAssignments$,
$phase := checkConstraints$
endif

Phase *checkConstraints* then comprises the evaluation of the considered constraints of $checkInv$, $checkPre$, and $checkPost$. We partition this phase as follows.

if $phase = checkConstraints$ **then**
$CheckInvariants$,
$CheckPreconditions$,
$CheckPostconditions$,
$phase := identifyViolations$
endif

The validity of OCL expressions is determined by an interpretation function $I[[\ ]]$ over so-called *environments* τ [11, Section A.3.1.2]. An environment $\tau = \langle \sigma(\mathcal{M}), \beta \rangle$ comprises the system state $\sigma(\mathcal{M})$ and an OCL-specific variable assignment β that maps variable names to values. Function β determines values for those variables that appear in OCL let expressions and as iterator variables of predefined collection operations.

Table 1. Noteworthy Changes for OCL Evaluation

$NewObjects := \{\underline{oid}_1, \ldots, \underline{oid}_n\}$, where $\underline{oid}_1, \ldots, \underline{oid}_n$ are the **objects** of classes $c_j \in CLASS, 1 \leq j \leq n$, that are newly **created**.
$DestroyedObjects := \{\underline{oid}_1, \ldots, \underline{oid}_m\}$, where $\underline{oid}_1, \ldots, \underline{oid}_m$ are the **objects** of classes $c_j \in CLASS, 1 \leq j \leq m$, that are **destroyed**.
$NewAttributeValues := \{a_1, \ldots, a_l\}$, where $a_1, \ldots, a_l$ are the **attributes** of objects $\underline{oid}_j, 1 \leq j \leq l$, whose values are **changed**.
$NewLinks := \{l_{as_1}, \ldots, l_{as_k}\}$, where $l_{as_1}, \ldots, l_{as_k}$ are the **links** of associations $as_j \in ASSOC, 1 \leq j \leq k$, that are newly **established**.
$DestroyedLinks := \{l_{as_1}, \ldots, l_{as_p}\}$, where $l_{as_1}, \ldots, l_{as_p}$ are the **links** of associations $as_j \in ASSOC, 1 \leq j \leq p$, that are **removed**.
$NewConfigurations := \{cfg1, \ldots, cfg_q\}$, where $cfg1, \ldots, cfg_q$ are the **new state configurations** that are **reached** for objects $\underline{oid}_j$ of active classes $c_j, 1 \leq j \leq q$.
$MessagesSent := \{opExec_1, \ldots, opExec_r\}$, where $opExec_j$ denotes an operation execution with name op_j and identifier $opId_j, 1 \leq j \leq r$, of objects $\underline{oid}_j$ from classes $c_j \in CLASS$ that **sent a message** named msg_j with identifier $msgId_j$ and actual parameter values $v_{j,1}, ..., v_{j,n_j}$ to object $destId_j$ with identifier $callId_j$.
$MessagesReceived := \{\underline{oid}_1, \ldots, \underline{oid}_v\}$, where $\underline{oid}_j$ denotes an object of class $c_j \in CLASS, 1 \leq j \leq v$, that **receives a message** named msg_j with call identifier $callId_j$ invoked by an operation execution of object $srcId_j$ of a class c'_j (where c'_j is identified by $srcOp_j$ and $srcOpId_j$).
$ConsumedSignals := \{sigSent_1, \ldots, sigSent_w\}$, where $sigSent_j = \langle srcId_j, srcOp_j, srcOpId_j, sig_j, callId_j \rangle, 1 \leq j \leq w$, is a **signal** that is **consumed** by objects $\underline{oid}_j$ of classes $c_j \in CLASS$.
$DispatchedOperations := \{opCalled_1, \ldots, opCalled_x\}$, where $opCalled_j = \langle srcId_j, srcOp_j, srcOpId_j, op_j, callId_j \rangle, 1 \leq j \leq x$, is a **waiting operation call** that is **dispatched** by objects $\underline{oid}_j$ of classes $c_j \in CLASS$.
$TerminatedOperations := \{opExec_1, \ldots, opExec_y\}$, where $opExec_j, 1 \leq j \leq y$, denotes an **operation execution** with name op_j and identifier $opId_j$ of an object $\underline{oid}_j$ that **terminated**.
$ReturnedOperations := \{opExec_1, \ldots, opExec_z\}$, where $opExec_j, 1 \leq j \leq z$, denotes a **terminated operation** with name op_j and identifier $opId_j$ of an object $\underline{oid}_j$ that **returns** with its result value $result$.

Note that we specified a macro *UpdateOclVariableAssignments* in the rule for *phase = initConstraintEvaluation* to indicate the update of β at trace position i. But as β is precisely defined in the formal OCL 2.0 semantics [11, Section A.3.1.1], we omit further details of the variable update here.

Generally, the semantics of an OCL expression $expr \in Expr_t$ of type t is a function $I[[expr]] : Env \rightarrow I(t)$ from the set Env of all environments to the semantic domain of t, i.e., $I(t)$. However, for postconditions *two* environments have to be considered, i.e., the current environment τ and the past environment τ_{pre} that represents the system state at the time of the start of the investigated operation execution.

As invariants, pre-, and postconditions are OCL expressions with Boolean result type, we here simply apply the expression interpretation function $I[[\ ]]$ as defined in the OCL 2.0 semantics [11, Section A.3.1]. However, note that we annotate function $I[[expr]]$ by obj to denote that $expr$ is evaluated w.r.t. that object.[3] The resulting rule for checking invariants is then defined by

$CheckInvariants \equiv$
  $\forall\ \langle obj, inv\rangle \in checkInv$:
    if $I[[inv]]_{obj}(\tau) = false$ **then**
      $violatedConstraints = violatedConstraints \cup \{\ \langle obj, inv\rangle\ \}$
    endif
    if $I[[inv]]_{obj}(\tau) = \bot$ **then**
      $undefinedConstraints = undefinedConstraints \cup \{\ \langle obj, inv\rangle\ \}$
    endif

As the rule for checking preconditions is defined in a very similar way, we here just present the more interesting rule for postconditions, as two system states have to be considered in this case:

$CheckPostconditions \equiv$
  $\forall\ \langle opExec, post\rangle \in checkPost$:
    choose $\tau_{pre} = \langle \sigma(\mathcal{M})_{[j]}, \beta_{[j]}\rangle$ **in** $TRACE$
        satisfying $\sigma(\mathcal{M})_{[j]}$ is the system state in which $opExec$ has started
      if $I[[post]]_{opExec}(\tau_{pre}, \tau) = false$ **then**
        $violatedConstraints = violatedConstraints \cup \{\ \langle obj, inv\rangle\ \}$
      endif
      if $I[[post]]_{opExec}(\tau_{pre}, \tau) = \bot$ **then**
        $undefinedConstraints = undefinedConstraints \cup \{\ \langle obj, inv\rangle\ \}$
      endif
    endchoose

Constraint Violations. Finally, we set the function *violation* to true if there is a violated constraint. We keep the violations in dedicated sets for further usage by the system, e.g., for a *violation report*. However, we do not define *what* has to be done if a constraint

[3] This is in contrast to the OCL semantics in which an invariant is always evaluated for *all objects* of the invariant's context class [11, Section A.3.1.5]. Our model of course also allows for this. However, we find it more flexible to also allow that only particular invariants are evaluated for particular objects.

fails. This is left to a user-defined mechanism in the system (e.g., exception handling, transaction rollback). That mechanism might benefit from our approach and check for a violation report at completion of the current evaluation cycle, i.e., when state *identifyViolations* is reached.

if $phase = identifyViolations$ **then**
  if $violatedConstraints \neq \emptyset \ \vee \ undefinedConstraints \neq \emptyset$ **then**
    $violation := true$
  endif ,
  $phase := updateSystemState$
endif

5 Related Work

Various works on the semantics of OCL have been published, ranging from early definitions for OCL version 1.1 back in 1998 to the most recent version OCL 2.0 in March 2004. A good overview of the work is given in [4]. However, none of the existing formal OCL semantics covers all language features. This is due to the fact that the standard leaves several issues open since they are still under investigation, e.g., non-determinism, empty collections, and recursive specifications.

Especially language features for checking activated states and sent messages have not received much attention. Although it is possible to refer to State Diagram states and check for activated states with operation `oclInState()` since OCL version 1.3, only the authors of this article have presented a formal integration of State Diagram states with OCL yet [6]. Concerning OCL messages, which was originally proposed in [8, 9], we only know of one other approach that deals with the corresponding formal semantics [10]. However, we are not aware of any other OCL semantics that supports the OCL message concept and precisely defines when OCL constraints have to be checked.

6 Conclusion

Our ASM definition of the semantics of OCL makes use of a flexible, generic approach that allows the executed model (or: the system) to trigger the evaluation of OCL constraints at arbitrary times. This is in accordance with the OCL language definition that deliberately leaves it open when to check OCL constraints, which is mainly due to performance aspects of the individual application. Our approach allows to rigorously check constraints at all relevant times, namely by synchronization of noteworthy changes with the evaluation trigger *evaluateConstraints*. For constraint evaluation in sequential implementations, the system has to be interrupted until the evaluation is completed in order to extract the evaluation result. This, however, is not very efficient for general application. Therefore, we see our approach as a prerequisite framework for modellers that want to precisely define *when* to check OCL constraints. Generally, there are a number of aspects that have an impact on this issue, e.g., the different phases of the development process or the specific application domain.

As a next step, we want to use our ASM definition as a basis to precisely identify and check which constraints have to be evaluated in what states. Our application domain is the modelling of time-critical manufacturing systems. For this, we also plan to combine the presented ASM definition with related definitions for State and Activity Diagrams.

Acknowledgements

This work receives funding through the DFG project GRASP within the DFG priority programme 1064 'Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen' and partial funding through the DFG Research Centre 614 'Selbstoptimierende Systeme des Maschinenbaus'.

References

[1] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *AMAST 2000, Iowa City, USA*, volume 1816 of *LNCS*, pages 293–308. Springer, 2000.

[2] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *Abstract State Machines, Theory and Applications (ASM 2000), Monte Verità, Switzerland*, volume 1912 of *LNCS*, pages 223–241. Springer, 2000.

[3] E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer, 2003.

[4] M. V. Cengarle and A. Knapp. OCL 1.4/1.5 vs. OCL 2.0 Expressions: Formal Semantics and Expressiveness. *Software and Systems Modeling (SoSyM)*, 3(1):9–30, March 2004.

[5] S. Flake. Towards the Completion of the Formal Semantics of OCL 2.0. In *27th Australasian Computer Science Conference (ACSC 2004), Dunedin, New Zealand*, pages 73–82, 2004.

[6] S. Flake and W. Müller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Software and System Modeling (SoSyM)*, 2(3), October 2003.

[7] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, UK, 1994.

[8] A. Kleppe and J. Warmer. Extending OCL to Include Actions. In *UML 2000 – Advancing the Standard. York, UK*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[9] A. Kleppe and J. Warmer. The Semantics of the OCL Action Clause. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL*, pages 213–227. Springer, 2002.

[10] M. Kyas and F. de Boer. On Message Specifications in OCL. In *UML 2003 Workshop on Compositional Verification of UML Models*, San Francisco, CA, USA, October 2003.

[11] OMG, Object Management Group. UML 2.0 OCL Specification. OMG Adopted Specification ptc/03-10-14, October 2003.

[12] OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.

[13] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, 2001.

[14] M. Stumptner and M. Schrefl. Behavior Consistent Inheritance in UML. In *19th Int. Conf. on Conceptual Modeling (ER 2000), Salt Lake City, UT, USA*, volume 1920 of *LNCS*, pages 527–542. Springer, 2000.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language – Getting your Models Ready for MDA*. Addison-Wesley Object Technology Series. Pearson Education, Inc., 2003.

[16] P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In J. Jürjens et al., editors, *Critical Systems Development with UML*, pages 53–62. Technische Universität München, Institut für Informatik, 2002.